Comments on *Impressions on MFC vs Qt Programming*

Written by Pascal Audoux
Translated and improved by Philippe Fremy

Personal

- Main page
- Some Photos
- Jean-Jacques Goldman
- Resume
- Cool stuff

Writings

- Main Page
- BLOG at advogato
- KDE has tea with RMS
- Paris Linux Solutions 2003
- Qt vs MFC
- KDE Developer Faq
- Paris Linux Expo 2002
- KPart Demonstration
- From Gtk to PyQt

Interviews

- Trolltech CEO (2004)
- KDevelop team
- KOffice team
- Guido van Rossum
- Fosdem organiser
- Trolltech CEO (2001)

Projects

After putting this article on the web, it has received the following critics :

- *It is not very well written*
- *MFC problems are not very well described*
- *There is no code examples*
- *Qt is praised all over the article, so the article is biased*
- *The author does not show some deep knowledge of MFC and states false things*
- *The author does not compare Qt with .NET*

I would like to respond: it is very hard and very time consuming to write an article. It is even more time consuming to write a good article with backed-up facts, code examples, comparisons, ... If I had such a good article, I probably would have published it on a more professional source, but not simply on my website.

I recognised these critics to be valid to some extent. This article was written to provide a slight overview of MFC programming but it reflects more all the problem we have faced when programming with MFC than a pure Qt/MFC comparison.

- Main Page
- Luaunit
- Keep Cool
- Yzis
- Indent Finder
- Pretty Make
- KVim
- QCppUnit
- Klotski
- Jayacard
- KMerge
- Good tools

However, I still think that the material presented thereafter is valuable. I havn't found any comparison of MFC and Qt programming sofar. So please read on the only one.

The critics are online here: comments on MFS vs Qt. Drop me a mail if you want to add something.

Philippe Fremy

Introduction

I have been programming in both Qt and MFC. I would like to share with you my experience of the differences between using the two toolkits.

I am not a professional writer. So this article certainly is not as slick and clean as something you would find in a professional website or magazine. This is just my experience, that I share with you, in my own words. English is not my native language, so my constructs are probably a bit strange and there are mistakes. Please mail them to me, so that I can fix them.

This article does not pretend to be objective. It is just a report of my personal experience. I do not address every good and bad point of Qt or MFC. The fact that I knew Qt programming before starting MFC programming might alter my objectiveness.

This article is written from a pragmatic point of view: My boss gives me the specification of the applications he wants

and I develop them. I have developed some with Qt, and some other with MFC.

The Microsoft Foundation Class (MFC) is a graphical toolkit for the windows operating system. It is a more or less object oriented wrapper of the win32 API, which causes the API to be sometimes C, sometimes C++, and usually an awful mix.

Qt is a graphical C++ toolkit started around 94 by Trolltech (www.trolltech.com). It runs on Windows (any version), Mac OS X, any Unix and on embedded devices such as the Sharp Zaurus. Qt is clearly and cleanly object oriented.

Document/View model

MFC programming requires the use of Document/View model and templates. It is almost impossible not to use them. However, templates have a fixed structure and it is very difficult to use them for something that was not planned by the template conceptor. Try for example to split an area and display two views on two different documents. Impossible. Another problem is that often, the template creates the view but it is not possible to access it. Everything must be done by the document and this can sometimes create problems.

Qt doesn't force any design model. You can use document/view without any problem, if you think it is

appropriate. But you can go without it.

Pseudo Object Design vs Good Object

The fundamental difference between Qt and MFC is their design.

MFC is a kind of object wrapper allowing access to the windows API, which is written in C. This is not a good object oriented design. In many places, you must supply a C struct with 15 members but only one relevant to your case, or you must call functions with obsolete parameters.

And there are nasty tricks, without any consistency. For example, if you create a graphical object, it is not created until the `Create()` methods has been called. For dialogs, you must wait for `OnInitDialog()`, for views, you wait for `OnInitialUpdate()`, ... So if you create an object manually and call its methods, your program crashes.

Let's take the example of a dialog containing a CEdit control. You can not use the method `GetWindowText()` on this field if you are not inside the `DoModal()` method. It will fail miserably. Why isn't it possible to fetch the text of a control when the control is not in a certain state ? MFC is full of these nasty tricks. And it is hard to debug.

Qt is the opposite. The

architecture is a good object oriented one and was obviously intelligently designed. The result is a toolkit very consistent regarding naming, inheritance, classes organisation and methods. Methods argument are the one you want to supply, no more. They always come in the same order for different classes. And the return value is logical. Everything is at the same time powerful and simple. Once you have used one of their classes, you can use many of them because they all work the same.

With Qt, to get an Edit control, you create your QLineEdit control with new, like any normal C++ class. You can immediately and forever access all its methods, whether it is shown or not. There is simply no trick, it works in the simplest way you could imagine.

Message loop

MFC is an event driven framework. To perform anything, you must react on certain messages. There are thousand messages sent by Windows to the program. Unfortunately, it is thus not easy to know which one to use, what information they contain, when they are exactly emitted, ... Some are redundant, some are not emitted, some are not documented. The documentation is not very good on this topic. It is difficult to know which objects emits what, when, which object receives or does not receive it and what action is taken. Some features available through

messages could perfectely be available through direct calls. This message stuff doesn't help debugging or code review.

Qt is based upon a powerful callback mechanism based on signal emission and reception inside slots. This system is the core communication mechanism between objects. It can passes any number of arguments within the signal. It is very powerful. You connect directly your relevant signals to your slots so you know what is going on. The number of signals emitted by a class is usually small (4 or 5) and is very well documented. You feel much more in control of what is going on. This signal/slot mechanism resemble the Java listener approach, except that it is more lightweight and versatile.

Interface Creation

MFC does not handle layout in windows: this creates problems when one wants a window with a variable size. You must reposition your controls by hand on resize requests (This explains why many windows dialog are not resizable). The problem gets bigger with applications that must be translated, because many languages express things with longer words or sentences. You must rearrange your windows for every language.

Visual Studio's resource editor is very limited: you can place controls at fixed positions and that's it. A few properties can be adjusted and the class wizard allow to create variables

wizard allow to create variables and methods easily. However, it is worth noticing that it would be very tedious to create these manually: the message loop, the DDX, the IMPLEMENT_DYNCREATE are far too complicated.

With Qt, everything can be coded manually because it is simple: to get a button, you just write:

```
button = new QPushButton
( "button text ",
MyParentWidget);
```

If you want a method action() to be executed when the button is clicked, you write:

```
connect( button, SIGNAL
( clicked() ), SLOT
( action() ) );
```

Qt has a powerful layout mechanism which is so simple that you waste time when you do not use it.

Qt provides a graphical tool, Qt Designer, to help building the interface. You can adjust any properties of the controls you use. You don't put them at fixed places, it uses layout to organise things nicely. You can connect signal and slots with this tool, so it does more than simple interface design. The tool generates code that you can actually read and understand. Because the generated code is in a separate file, you can regenerate your interface as many times as you want, while coding at the same time.

Qt designer lets you do things that are not possible in MFC,

like creating a listview with pre-filled fields, or using a tab control with different views on each tab.

Documenteation - Help

Documentation is an important consideration when one wants to use a rich graphical toolkit. Visual's documentation, MSDN (for which you must pay separately) is huge, on 10 CDROM. It features many articles, that cover many tasks. However, it gives the feeling that it documents poor design choices and misfeatures, rather than current useful feature. The cross-linking is also very poor. It is difficult to go from a class to its parent or child class, or to related classes. Methods are presented without their signature. It is difficult to access inherited methods of a class. Another problem is that if look for a keyword, you'll get help on this keyword on VC++, Visual Basic, Visual J++, InterDev, even if you filter your results.

Qt's documentation is excellent. The best is to look by yourself: doc.trolltech.com

It is complete and copious in the sense that it covers every Qt area, but fits in 18 Mbytes. Every class and methods is properly documented with plenty of details and examples. It is easy to navigate from classes and methods to other classes, using either the html version or the tool provided by Trolltech (Qt Assistant). The documentation comes with a tutorial and example of typical

tutorial and example of typical usage. There is also a FAQ and a public mailing list, accessible via a newsgroup or searchable web interface. If you have a license, you can ask the support which responds usually within one day.

The fact that Qt is well thought out helps a lot to reduce the need for external help. One of the stated goal of Trolltech is "The product and the documentation should be so good that no support is needed".

Unicode

With MFC, to get unicode display, you must compile and link with specific options (and change the entry point of the executable). The you must add _T around every strings that you use in your program. You must change all your 'char' to TCHAR. Every string manipulation function (strcpy, strdup, strcat, ...) is replaced with other functions with a different name. And the most annoying is that a program compiled with unicode support won't work with a DLL compiled without unicode support. This is very problematic when you develop with external DLL, on which you have no control.

With Qt, strings are handled in a class QString that is natively unicode. No compilation/link requirements, no code alteration, just QString. The Qt code is natively unicode and there is no problem.

The QString class itself is very

powerful so you use it everywhere, even when you don't care about unicode. This makes transition to unicode very easy. QString provides conversion functions to `char *` or UTF8 if required.

Technically, the big difference comes from the design of the MFC class CString, to be compared with the QString. CString is basically a `char *` with a few methods. The advantage is that everywhere where you need `char *`, you can use CString member. It looks good at first glance, but this has big drawbacks, specifically because you can modify directly the `char *` of the CString without updating the class. This is also a problem when converting to Unicode.

QString, on the contrary, stores internally a unicode version of the string, and provides a `char *` only when required. The whole Qt api requires QString for text arguments, so you very seldom use `char *`. The QString has also some additional facilities, like automatic sharing (or lazy copy) of the content of the string. The result is a very powerful class that you want to use everywhere. This is a typical example of a good design on the Qt side and a C hack wrapped in C++ on the MFC side.

Internationalisation

It is possible to internationalise a MFC program. Just put every string in a string table and use `LoadString( IDENTIFIER )` everywhere in your code. Then

every where in your code. Then, you must transform the resources into a DLL, translate the strings of the string table (using visual) into the desired language, translate the graphical resources (because their text can not be put into the string table) and ask the program to use this DLL. This is so complex that you probably can not defer it to a translator alone, he must be assisted. You will also have problems because in MFC, controls have a fixed position that depends on the non translated text. Longer translated strings will overlap. When you change some strings or add new strings, you must ensure manually that the translation has been updated.

With Qt, you just put your strings inside the `tr()` function . This is very lightweight when developing. You can change the reference strings directly in your code. A special program, Qt Linguist, will extract all the strings to be translated and display them with a friendly interface. The interface allow easy translation, with facilities such as use of a dictionnary, display of the context of the string, proper unicode display, shortcuts conflicts detection, detection of new untranslated strings, detection of modified strings. This program can be used by a translator with no development knowledge. The editor is available under GPL so you can modify it. The translated file is in XML, so it can even be easily reused in a different context. Adding a new translation to an application is a matter of producing a new file

with Qt linguist.

Resources problem

With MFC, a part of the development process depends on "resources". You need them for many cases. This has consequences:

- It is almost impossible to develop with another tool than Visual Studio
- The resource editor has limited features. For example, it is not possible to ajust everything with the dialog editor: some properties can be changed and other not. For toolbars, you are forced to use an image that contains the images of all the buttons. To set the application name, you must put certain strings in the string table, but you must know that every field is separated by '\n'. The order and the signification of every field is neither obvious nor easy to find.
- Resources are mapped to #defined numbers in the file "Resource.h" (a number < 32768). This creates problem when deleting or renaming resources. It is also a nightmare when many DLL use resource.h files with the same resources name but different numbers (typical case of a framework with DLL components).

- When using a DLL with its own resources, but which uses other DLL, there are many chances that the program mixes the resources of the program and the DLLs (even with #define to the same values). You must then reserve exclusive ranges for the resources, but it is not always possible as you don't necessary have control on any DLL.

With Qt, there is no concept of resources, which solves all the mentionned problems. Qt provides a small script to include images into your code. For interface creation, three is Qt Designer that generates readable code.

Price

Once you have bought Visual Studio, you get MFC SDK for free.

Qt is free in its Unix version (available under GPL) for Free Software. A non commercial version is available on Windows. But for commercial close source development, you must pay a Qt license. The license is for one platform (Unix, MacOs or Windows), per developer. It must be bought once forever for every developer and a one year support is included. There is no runtime distribution fee. The price is quite high for a small company: 1550 $ (there are discount for more than one license). Note that the cost is less than half a month of a

developer. If you compare your development cost with MFC and Qt, Qt will make you earn far more than half a month in time of development and feature completeness. The investment is worth it.

Distribution

To distribute your MFC application, you could rely on MFC being present on Windows. But this is not very safe. Under the same name, MFC42.dll, you can get three versions of the same library. You usually have to check that the user has the correct version of MFC42.dll, and else upgrade it. Upgrading MFC alters the behavious of many applications. This is not something I am comfortable with. What if the customer PC stops working after installing my program ?

Qt names its DLL explicitely (qt-mt303.dll) so there is no risk of altering the behaviour of an application depending on, let's say qt-203.dll, when installing qt-mt303.dll . There is also no "I update your whole system" issue.

Other advantages of Qt

Qt is not only a concurrent of MFC. It is a full toolkit, with many features available in a simpler way than in MFC, and many that simply have no equivalent in MFC:

- **Controls**: Qt is a graphical toolkit, so provides a rich set of

provides a rich set of graphical controls: labels, combo box, toggle buttons, ... Some of them are very sophisticated, like the listview which allow to have multi column list view with icons and toggle buttons.

- **XML**: Qt provides classes to manipulate XML documents (using Sax2 or Dom). The tool is simple, powerful, complete and bug free.

- **Regular Expressions:** Qt provides full support for perl-compatible regular expression. This goes far beyound the simple '?' and '*' meta-characters. Regular Expressions are a very powerful tool, to parse documents, to grep patterns inside documents, to build filters, to define masks for edit controls.

- **Multi-platform:** Qt 3 is multi-platform. It works on Windows (any of them), Unix (any of them), MacOs X and embedded platforms. You just have to recompile your code to get it working on a different platform. Except for the compiler adjustments (or limitations), you don't have to touch your code.

- **Template classes:** Qt provides useful classes to handle lists, files, dictionnaries, strings,

threads, ... All of them are very powerful and very handy; more than the STL and the MFC equivalents.

- **Memory management:** Qt has many facilities that makes memory management a no-brainer. Qt objects are automatically destroyed when their parent is destroyed. Many classes have an implicit sharing mechanism that relieves you from the pain of managing destruction and copy of these objects.

- **Network API:** Qt features classes to help programming with Network programming: socket, dns, ftp, http, ...

- **Database API:** Qt features classes for seamless database integration : Data aware wigets, database connection, SQL queries, ...

- **OpenGL API:** Qt provides classes for seamless integration with OpenGL (3D accelearted) libraries.

- **Canvas:** Qt provides classes optimised for handling quickly moving 2d objects, usually known as sprites.

- **Styles:** It is possible to fully customize the rendering of all the Qt controls. That way, Qt emulates the style of all

emulates the style of all the available toolkits: Motif, MFC, NextStep, ...

What about Codejock ?

The many drawbacks of MFC have left room for companies to sell MFC wrappers, which help to actually build applications easely. We have been using the CodeJock library. How does CodeJock + MFC compares to Qt ?

- CodeJock is a wrapper around MFC which is a wrapper around the windows API. Adding more wrappers to hide problems is usually not a good solution. All the cited problems still exist (resources, templates for doc/view, messages, unicode, internationalisation, ...)

- The classes provided by CodeJock allow easier use of MFC controles (simpler methods or more methods, added features). It it then possible for example to create a multi-tab view while it is _Mission Impossible_ with MFC. However, the library is very limited, providing only a few more classes than MFC, not a full set. We are closer to the set of patches than to a wrapper.

- The quality of the library is poor. Many bugs are left. new are added.

During the first 6 month of 2002, there was 3 releases (1.9.2.1, 1.9.2.2, 1.9.3.0), every of them correcting more than fifty bugs, including major ones. The library is actually neither stable nor tested. This is not a professional quality tool. Users are alpha testers. Also note that the API changes between releases, and you must sometime alter your own code to adapt the new versions. This is a hint of the poor design.

- Reading the code (unfortunately unavoidable for codejock, given certain strange behaviours) reveals tons of horrors: methods with more than 500 lines, with some redundant code and plenty of return in the middle of nowhere, very few comments and many many hacks. Many classes members are declared public where they should indeed be protected and so on.

- Documentation is sparse, or void in certain cases (the method is present in the documentation, with no explanation of anything). Documenting doesn't look like a priority given its absence of progress in the last releases.

- There are no features present in CodeJock that you can not find in Qt.

Except for the hexadecimal editor, which unfortunately is buggy as hell and that you can easily do in Qt (examples already exist).

Qt's code quality is very good. The library is always stable and robust. During the last 6 years, the source and binary compatibility was broken only twice, to add major features. And only in once case (Qt1 to Qt2) would this break require substantial code alteration.

Conclusion

The conclusion drawn from our personal experience is obvious. Qt is far better than MFC. You'll produce better programs with less hassle.

Some people complained that this article is biased toward Qt and does not present any MFC advantage. This is simply our experience : we had tons of problems with MFC, and almost none with Qt. Programming with Qt was always simple, documented and efficient. If MFC has good points, we have not found them, apart from being delivered free with Visual Studio.

We are of course open to feedback: for suggestions, improvements, remarks and flames, mail us!

I would like to include quotes of people who have used both MFC and Qt. If you have done so, please drop me a mail.

Links

Other material comparing Qt:

- From Gtk to PyQt, by Philippe Fremy: analyses the same program written in Gtk, Qt and PyQt.
- Qt vs Java whitepaper, by Mathias Kalle Dallheimer.
- Guillaume Laurent explains why Qt is better than gtkmm
- Gui Framework

Last modification : $Date: 2005/01/30 07:04:17 $ - $Author: philippe $